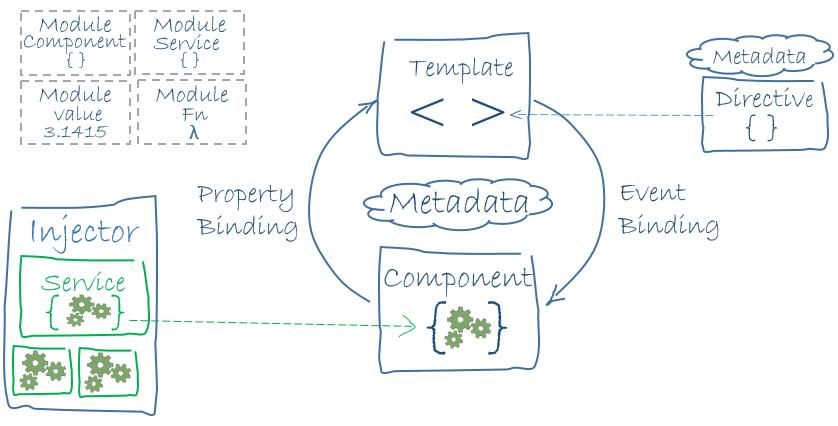
# Architecture

ith Angular, we write applications by composing HTML templates with Angularized-markup, writing component classes to manage those templates, adding application logic in services, and handing the top root component to Angular's bootstrapper.



## Modules

Angular apps are modular.In general we assemble our application from many modules.

app.component.ts

export class AppComponent{…}

The export statement tells TypeScript that this is a module whose AppComponent class is public and accessible to other modules of the application.

### Import from local modules

When we need a reference to the AppComponent, we **import** it like this:

app/main.ts (import)

import{AppComponent}from'./app.component';

The import statement tells the system it can get an AppComponent from a module named app.component located in a neighboring file. The module name (AKA module id) is often the same as the filename without its extension.

### Import from Libraries

Some modules are libraries of other modules. Angular itself ships as a collection of library modules within several npm packages. Their names begin with the @angular prefix.

app/app.component.ts (import)

import{Component}from'@angular/core';

## Component

A component controls a patch of screen real estate that we could call a view. The shell at the application root with navigation links, that list of heroes, the hero editor ... they're all views controlled by components.

We define a component's application logic — what it does to support the view — inside a class. The class interacts with the view through an API of properties and methods.

A Component has a lifecycle managed by Angular itself. Angular creates it, renders it, creates and renders its children, checks it when its data-bound properties change, and destroys it before removing it from the DOM.

Directive and component instances have a lifecycle as Angular creates, updates, and destroys them.

|  |  |
| --- | --- |
| **Hook** | **Purpose** |
| ngOnInit | Initialize the directive/component after Angular initializes the data-bound input properties. |
| ngOnChanges | Respond after Angular sets a data-bound input property. The method receives achanges object of current and previous values. |
| ngDoCheck | Detect and act upon changes that Angular can or won't detect on its own. Called every change detection run. |
| ngOnDestroy | Cleanup just before Angular destroys the directive/component. Unsubscribe observables and detach event handlers to avoid memory leaks. |

**Components only:**

|  |  |
| --- | --- |
| **Hook** | **Purpose** |
| ngAfterContentInit | After Angular projects external content into its view. |
| ngAfterContentChecked | After Angular checks the bindings of the external content that it projected into its view. |
| ngAfterViewInit | After Angular creates the component's view(s). |
| ngAfterViewChecked | After Angular checks the bindings of the component's view(s). |

### Lifecycle Sequence

Angular calls the lifecycle hook methods in the following sequence at specific moments:

|  |  |
| --- | --- |
| **Hook** | **Timing** |
| ngOnChanges | before ngOnInit and when a data-bound input property value changes. |
| ngOnInit | after the first ngOnChanges. |
| ngDoCheck | during every Angular change detection cycle. |
| ngAfterContentInit | after projecting content into the component. |
| ngAfterContentChecked | after every check of projected component content. |
| ngAfterViewInit | after initializing the component's views and child views. |
| ngAfterViewChecked | after every check of the component's views and child views. |
| ngOnDestroy | just before Angular destroys the directive/component. |

The router component, for instance, has it’s own router lifecycle hooksthat allow us to tap into specific moments in route navigation.

## Metadata

Metadata tells Angular how to process a class.In TypeScript, we attach metadata by using a decorator. Here's some metadata for HeroListComponent:

@Component({

selector: 'hero-list',

templateUrl: 'app/hero-list.component.html',

directives: [HeroDetailComponent],

providers: [HeroService]

})

export class HeroListComponent implements OnInit {

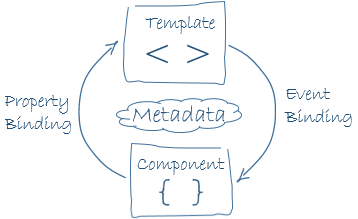
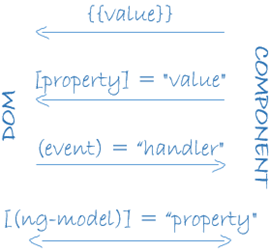
/\* . . . \*/

}

**A decorator is a function.** Decorators often have a configuration parameter. The @Component decorator takes a required configuration object with the information Angular needs to create and present the component and its view.

* directives: array of the components or directives that this template requires.We saw in the last line of our template that we expect Angular to insert a HeroDetailComponent in the space indicated by <hero-detail> tags. Angular will do so only if we mention the HeroDetailComponent in this directives array.
* provides: array of **dependency injection providers** for services that the component requires. This is one way to tell Angular that our component's constructor requires a HeroService so it can get the list of heroes to display.

## Data binding



<li>{{hero.name}}</li>

<hero-detail [hero]="selectedHero"></hero-detail>

<li (click)="selectHero(hero)"></li>

* Interpolation: The {{hero.name}} interpolation displays the component's hero.name property value within the <li> tags.
* Property binding: The [hero] property binding passes the value of selectedHero from the parent HeroListComponent to the hero property of the child HeroDetailComponent.
* Event binding:The (click) event binding calls the component's selectHero method when the user clicks a hero's name.
* Two-way data binding: using the ngModel directive.

## Services

Service is a broad category encompassing any value, function, or feature that our application needs.

We prefer our component classes lean. Our components don't fetch data from the server, they don't validate user input, and they don't log directly to the console. They delegate such tasks to services.

Angular doesn't enforce these principles. It won't complain if we write a "kitchen sink" component with 3000 lines.

Angular does help us follow these principles by making it easy to factor our application logic into services and make those services available to components through dependency injection.

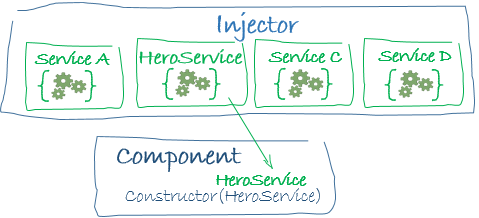
## Dependency Injection

Most dependencies are services. Angular uses dependency injection to provide new components with the services they need.

Angular can tell which services a component needs by looking at the types of its constructor parameters. For example, the constructor of our HeroListComponent needs a HeroService:

app/hero-list.component.ts (constructor)

constructor(private service:HeroService){}



# Displaying Data

## Showing component properties with interpolation

@Component({  
**selector**: **'my-app'**,  
**template**: **`  
<h1>{{title}}</h1>  
<h2>My favorite hero is: {{myHero}}</h2>  
 `**})  
*// More compact compare with constructor***export class** AppComponent {  
**title** = **'Tour of Heroes'**;  
**myHero** = **'Windstorm'**;  
}  
*// or initialize variable in constructor***export class** AppCtorComponent {  
**title**: **string**;  
**myHero**: **string**;  
**constructor**() {  
**this**.**title** = **'Tour of Heroes'**;  
**this**.**myHero** = **'Windstorm'**;  
 }  
}

If a property is showing in the view, Changing the value of a property will trigger a corresponding update in the view, the showing value will be updated.

@Component({  
**selector**: **'my-app'**,  
**template**: **'<h1>My First Angular 2 App</h1><h2>It is the {{*name*}}</h2>'**})  
**export class** AppComponent {  
**name**: **string** =**"hello"**;  
**constructor**(){  
*setTimeout*(()=>{  
**this**.**name** = **"hello world!"**; *// after 5 seconds showing new value: "hello world!".*}, 5000);  
 }  
}

## Showing an array prpoerty with \*ngFor

@Component({  
**selector**:**'my-app'**,  
**template**: **`   
<h1>{{title}}</h1>  
<h2>My favorite hero is: {{myHero}}</h2>  
<p>Heroes:</p>  
<ul>  
<li \*ngFor="let hero of heroes">  
 {{ hero }}  
</li>  
</ul>`**})  
**export class** AppComponent {  
**title** = **'Tour of Heroes'**;  
**heroes** = [**'Windstorm'**, **'Bombasto'**, **'Magneta'**, **'Tornado'**];  
**myHero** = **this**.**heroes**[0];  
}

## Conditional display with \*ngIf

**<p \*ngIf="heroes.length> 3">There are many heroes!</p>**

Angular isn't showing and hiding the message. It is adding and removing the paragraph element from the DOM. That hardly matters here. But it would matter a great deal, from a performance perspective, if we were conditionally including or excluding a big chunk of HTML with many data bindings.

# User Input

User input triggers DOM events. We listen to those events with event bindings that funnel updated values back into our components and models.

When the user clicks a link, pushes a button, or enters text we want to know about it. These user actions all raise DOM events.he syntax is simple. We surround the DOM event name in parentheses and assign a quoted template statement to it.

**<button (click)="onClickMe()">Click me!</button>**

The (click) to the left of the equal sign identifies the button's click event as the **target of the binding**. The text within quotes on the right is the **template statement** in which we respond to the click event by calling the component's onClickMe method.A [template statement](#_Template_Statements)is a subset of JavaScript with restrictions and a few added tricks.

When writing a binding we must be aware of a template statement's **execution context**.That is usually the Angular component that controls the template .

@Component({  
**selector**: **'click-me'**,  
**template**: **`<button (click)="onClickMe()">Click me!</button>{{clickMessage}}`**})  
**export class** ClickMeComponent {  
**clickMessage** = **''**;  
onClickMe() {  
**this**.**clickMessage** = **'You are my hero!'**;  
 }  
}

When the user clicks the button, Angular calls the component's onClickMe method.

## Get user input from the $event object

**template**: **`  
<input (keyup)="onKey($event)">  
<p>{{***values***}}</p>  
`**

Angular makes an event object available in the $event variable, which we pass to the component's onKey() method.

**export class** KeyUpComponent\_v1 {  
**values** = **''**;  
*// without strong typing*onKey(event:**any**) {  
**this**.**values** += event.**target**.**value** + **' | '**;  
 }  
}

he shape of the $event object is determined by whatever raises the event. The keyup event comes from the DOM, so $event must be a standard DOM event object. The $event.target gives us in this example an HTMLInputElement, which has a **value** property that contains our user input data.

We generally prefer the strong typing that TypeScript affords. We can rewrite the method, casting to HTML DOM objects like this.

**export class** KeyUpComponent\_v1 {  
**values** = **''**;  
  
*// with strong typing*onKey(event: KeyboardEvent) {  
**this**.**values** += (<HTMLInputElement>event.**target**).**value** + **' | '**;  
 }  
}

Strong typing reveals a serious problem with passing a DOM event into the method: too much awareness of template details, too little separation of concerns.

## Get user input from a template reference variable

Here’s an example of using a template reference variable to implement a clever keystroke loopback in an ultra-simple template:

@Component({  
**selector**: **'loop-back'**,  
**template**: **`  
<input #box (keyup)="0">  
<p>{{*box*.value}}</p>  
 `**})  
**export class** LoopbackComponent { }

**But this won't work at all unless we bind to an event.**

Angular only updates the bindings (and therefore the screen) if we do something in response to asynchronous events such as keystrokes.

That's why we bind the keyup event to a statement that does ... well, nothing. We're binding to the number 0, the shortest statement we can think of. That is all it takes to keep Angular happy. We said it would be clever!

@Component({  
**selector**: **'key-up2'**,  
**template**: **`  
<input #box (keyup)="onKey(*box*.value)">  
<p>{{values}}</p>  
 `**})  
**export class** KeyUpComponent\_v2 {  
**values** = **''**;  
onKey(value: **string**) {  
**this**.**values** += value + **' | '**;  
 }  
}

An especially nice aspect of this approach is that our component code gets clean data values from the view. It no longer requires knowledge of the $event and its structure.

## Key event Filtering (with key.event)

Perhaps we don't care about every keystroke. Maybe we're only interested in the input box value when the user presses Enter, and we'd like to ignore all other keys.

When we bind to the (keyup) event, our event handling statement hears every keystroke. We could filter the keys first, examining every $event.keyCode, and update the values property only if the key is Enter.

Angular can filter the key events for us. Angular has a special syntax for keyboard events. We can listen for just the Enter key by binding to Angular's keyup.enter pseudo-event.

@Component({  
**selector**: **'key-up3'**,  
**template**: **`  
<input #box (keyup.enter)="values=*box*.value">  
<p>{{values}}</p>  
 `**})  
**export class** KeyUpComponent\_v3 {  
**values** = **''**;  
}

## On blur

@Component({  
**selector**: **'key-up4'**,  
**template**: **`  
<input #box  
(keyup.enter)="values=*box*.value"  
 (blur)="values=*box*.value">  
  
<p>{{values}}</p>  
 `**})  
**export class** KeyUpComponent\_v4 {  
**values** = **''**;  
}

# Template Syntax

## Template Statements

A template statement responds to an event raised by a binding target such as an **element**, **component**, or **directive**.

(<event>)="<statement>"

A template statement has a side effect. It's how we update application state from user input. There would be no point to responding to an event otherwise.

Like template expressions, template statements use a language that looks like JavaScript. The template statement parser is different than the template expression parser and specifically supports both basic assignment (=) and chaining expressions (with ; or ,).

However, certain JavaScript syntax is not allowed:

* New
* increment and decrement operators, ++ and –
* operator assignment, such as += and -=
* the bitwise operators | and &
* the template expression operators

As with expressions, statements can refer only to what's in the statement context — typically the **component instance** to which we're binding the event.

Template statements cannot refer to anything in the global namespace. They can’t refer to window or document.

The statement context may include an object other than the component. E.g. a [template reference variable](#_Template_reference_variables).

## Template reference variables

A template reference variable is a reference to a DOM element or directive within a template.

We can reference a template reference variable on the same element, on a sibling element, or on any child elements.

Here are two other examples of creating and consuming a Template reference variable:

*<!-- phone refers to the input element; pass its `value` to an event handler -->*<**input #phone placeholder="phone number"**>  
<**button (click)="callPhone(*phone*.value)"**>Call</**button**>  
  
*<!-- fax refers to the input element; pass its `value` to an event handler -->*<**input ref-fax placeholder="fax number"**>  
<**button (click)="callFax(fax.value)"**>Fax</**button**>

The hash (#) prefix to "phone" means that we're defining a phone variable.Folks who don't like using the # character can use its canonical alternative, the ref- prefix. For example, we can declare the our phone variable using either #phone or ref-phone.

Another example: ngForm and template reference variables. The HTML for a form can be quite involved.

<**form (ngSubmit)="onSubmit(*theForm*)" #theForm="ngForm"**>  
<**div class="form-group"**>  
<**label for="name"**>Name</**label**>  
<**input class="form-control" required ngControl="firstName"  
[(ngModel)]="currentHero.firstName"**>  
</**div**>  
<**button type="submit" [disabled]="!*theForm*.form.valid"**>Submit</**button**>  
</**form**>

What is the value of theForm?

It would be the HTMLFormElement if Angular hadn't taken it over. It's actually ngForm, a reference to the Angular built-in NgForm directive that wraps the native HTMLFormElement and endows it with additional superpowers such as the ability to track the validity of user input.

This explains how we can disable the submit button by checking theForm.form.valid and pass an object with rich information to the parent component's onSubmit method.

# Forms

## binding

If you use input-tag inside of a form, the name attribut must be defined.This is a requirement when using [(ngModel)] in combination with a form, so that we can easily refer to it in the aggregate form value and validity state.

<**input id="name" type="text" class="form-control" required [(ngModel)]="model.name"**>

The following error is shown:

ORIGINAL EXCEPTION: If ngModel is used within a form tag, either the name attribute must be set or the form control must be defined as 'standalone' in ngModelOptions.

<**input id="name" type="text" class="form-control" required [(ngModel)]="model.name" name="name"**>

The punctuation in the binding syntax**, [()]**, is a good clue to what's going on.

In a Property Binding, a value flows from the model to a target property on screen. We identify that target property by surrounding its name in brackets, []. This is a **one-way data binding from the model to the view**.

In an Event Binding, we flow the value from the target property on screen to the model. We identify that target property by surrounding its name in parentheses, (). This is a one-way data binding in **the opposite direction from the view to the model**.

No wonder Angular chose to combine the punctuation as [()] to signify a two-way data binding and a flow of data in both directions.

In fact, we can break the NgModel binding into its two separate modes as we do in this re-write of the "Name" <input> binding:

**<input type="text" class="form-control" required [ngModel]="model.name" (ngModelChange)="model.name = $event" >**

The **ngModelChange** is not an <input> element event. It is actually an event property of the NgModel directive. When Angular sees a binding target in the form [(x)], it expects the x directive to have an **x** input property and an **xChange** output property.

The other oddity is the template expression, model.name = $event. We're used to seeing an $event object coming from a DOM event. The ngModelChange property doesn't produce a DOM event; it's an Angular EventEmitter property that returns the input box value when it fires — which is precisely what we should assign to the model's name property.

We almost always prefer [(ngModel)]. We might split the binding if we had to do something special in the event handling such as debounce or throttle the key strokes.

## Validation

A form isn't just about data binding. We'd also like to know the state of the controls on our form.Using ngModel in a form gives us more than just two way data binding. It also tells us if the user touched the control, if the value changed, or if the value became invalid.

**ngModel** doesn't just track state; it updates the control with special Angular CSS classes from the set we listed above. We can leverage those class names to change the appearance of the control and make messages appear or disappear.

Internally Angular creates FormControls and registers them with an NgForm directive that Angular attached to the <form> tag. Each FormControl is registered under the name we assigned to the name attribute.

The NgModel directive doesn't just track state. It updates the control with three classes that reflect the state.

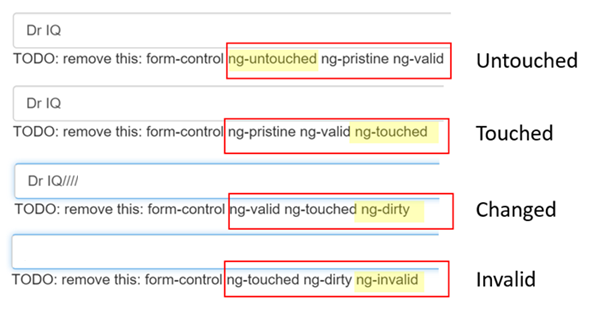
|  |  |  |
| --- | --- | --- |
| **State** | **Class if true** | **Class if false** |
| Control has been visited | ng-touched | ng-untouched |
| Control's value has changed | ng-dirty | ng-pristine |
| Control's value is valid | ng-valid | ng-invalid |

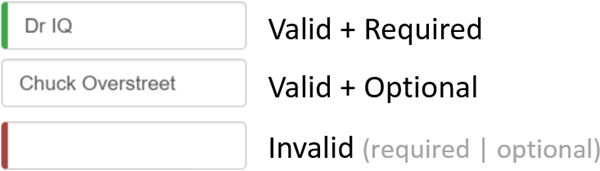
<inputtype="text"class="form-control"required

[(ngModel)]="model.name"

name="name" #spy>

<br>TODO: remove this: {{spy.className}}





We achieve this effect by adding two styles to a new forms.cssfile that we add to our project as a sibling to index.html.

forms.css

.ng-valid[required]{

border-left:5px solid #42A948; /\* green \*/

}

.ng-invalid {

border-left:5px solid #a94442; /\* red \*/

}

## Show and Hide Validation Error messages

We need a template reference variable to access the input box's Angular control from within the template. Here we created a variable called name and gave it the value "ngModel".

<**label for="name"**>Name</**label**>  
<**input type="text" class="form-control" required  
 [(ngModel)]="model.name"  
name="name" #name="ngModel"**>  
<**div [hidden]="*name*.valid || *name*.pristine" class="alert alert-danger"**>  
 Name is required  
</**div**>

We need a template reference variable to access the input box's Angular control from within the template. Here we created a variable called **name** and gave it the value "**ngModel**".

<**input id="name" type="text" class="form-control" required [(ngModel)]="model.name" name="name" #name="ngMod"**>

There is no directive with "exportAs" set to "ngMod" (" <input id="name" type="text" class="form-control" required [(ngModel)]="model.name" name="name" [ERROR ->]#name="ngMod">

Why "ngModel"? A directive's exportAs property tells Angular how to link the reference variable to the directive. We set name to ngModel because the ngModel directive's exportAs property happens to be "ngModel".

<div [hidden]="name.valid || name.pristine"class="alert alert-danger">

In this example, we hide the message when the control is valid or pristine; pristine means the user hasn't changed the value since it was displayed in this form.

<**form (ngSubmit)="onSubmit()" #heroForm="ngForm"**>

**ngSubmit** is an Angular directive and bind it to HeroFormComponent.onSubmit method with an event binding.

We defined a template reference variable, **#heroForm**, and initialized it with the value, "ngForm". The variable **heroForm** is now a reference to the **NgFormdirective** that governs the form as a whole.

What **NgForm** directive? We didn't add an NgForm directive! Angular did. Angular creates and attaches an NgForm directive to the <form> tag automatically.

The NgForm directive supplements the form element with additional features.

* It holds the controls we created for the elements with ngModel directive and name attribute and monitors their properties including their validity.
* It also has its own valid property which is true only if every contained control is valid.

<**button type="submit" class="btn btn-default"[disabled]="!*heroForm*.form.valid"**>Submit</**button**>

**Toggle two form regions:**

<div [hidden]="submitted">

<h1>Hero Form</h1>

<form \*ngIf="active" (ngSubmit)="onSubmit()" #heroForm="ngForm">

<!-- ... all of the form ... -->

</form>

</div>

submitted = false;

onSubmit() { this.submitted = true; }

When we click the Submit button, the submitted flag becomes true and the form disappears as planned.

# Dependency Injection

We don't have to create an Angular injector. Angular creates an application-wide injector for us during the bootstrap process.

bootstrap(AppComponent);

We do have to configure the injector by registering the providers that create the services our application requires. Provider can be registrated during bootstrapping:

bootstrap(AppComponent, [HeroService]); *// DISCOURAGED (but works)*

The injector now knows about our HeroService. An instance of our HeroService will be available for injection across our entire application. But it is discuraged. The bootstrap provider option is intended for configuring and overriding Angular's own preregistered services, such as its routing support.

The preferred approach is to register application providers in application components.

**import** {Component} **from '@angular/core'**;  
**import** {HeroListComponent} **from './hero-list.component'**;  
**import** {HeroService} **from "./hero.service"**;  
@Component({  
**selector**: **'my-heroes'**,  
**template**: **`  
<h2>Heroes</h2>  
<hero-list></hero-list>  
 `**,  
**providers**:[HeroService],  
**directives**: [HeroListComponent]  
})  
**export class** HeroesComponent {}

An instance of the HeroService is now available for injection in this HeroesComponent and all of its child components (e.g. HeroListComponent).

The HeroListComponent should get heroes from the injected HeroService. Per the dependency injection pattern, the component must ask for the service in its constructor.

**import** { Component } **from '@angular/core'**;  
**import** {Hero} **from "./hero"**;  
**import** {HeroService} **from "./hero.service"**;  
@Component({  
**selector**: **'hero-list'**,  
**template**: **`  
<div \*ngFor="let hero of heroes">  
 {{hero.id}} - {{hero.name}}  
</div>  
 `**})  
**export class** HeroListComponent {  
**heroes**: Hero[];  
**constructor**(heroService:HeroService){  
**this**.**heroes** = heroService.getHeroes();  
 }  
}

The constructor parameter type, the @Component decorator, and the parent's providers information combine to tell the Angular injector to inject an instance of HeroService whenever it creates a new HeroListComponent.

Dependencies are singletons within the scope of an injector. In our example, a single HeroService instance is shared among the HeroesComponent and its HeroListComponent children.

However, Angular DI is an [hierarchical injection](#_Hierarchial_Injectors) system, which means that nested injectors can create their own service instances.

**Testing the component:**

**let** expectedHeroes = [{**name**: **'A'**}, {**name**: **'B'**}]  
**let** mockService = <HeroService> {getHeroes: () =>expectedHeroes }  
  
*it*(**'should have heroes when HeroListComponent created'**, () => {  
**let** hlc = **new** HeroListComponent(mockService);  
*expect*(hlc.**heroes**.**length**).toEqual(expectedHeroes.**length**);  
});

**Service needs a Service:**

We create a logger service and register it with the aplication

**import** {Injectable} **from "@angular/core"**;  
@Injectable()  
**export class** Logger{  
**logs**:**string**[] = []; *// capture logs for testing*log(message: **string**) {  
**this**.**logs**.push(message);  
***console***.log(message);  
 }  
}

**Injects Logger direct in HeroService**

**import** { Injectable } **from '@angular/core'**;  
**import** { HEROES } **from './mock-heroes'**;  
**import** { Logger } **from './logger.service'**;  
@Injectable()  
**export class** HeroService {  
**constructor**(**private** logger: Logger) { }  
getHeroes() {  
**this**.logger.log(**'Getting heroes ...'**);  
**return** HEROES;  
 }  
}

The constructor now asks for an injected instance of a Logger and stores it in a private property called logger. We call that property within our getHeroes method when anyone asks for heroes.

@Injectable() marks a class as available to an injector for instantiation. Generally speaking, an injector will report an error when trying to instantiate a class that is NOT marked as @Injectable().

As it happens, we could have omitted @Injectable() from our first version of HeroService because it had no injected parameters. But we must have it now that our service has an injected dependency. We need it because Angular requires constructor parameter metadata in order to inject a Logger.

**SUGGESTION**: *add @Injectable() to every Service class. Always write @Injectable(), not just @Injectable. Our application will fail mysteriously if we forget the parentheses.*

Injectors are also responsible for instantiating components like HeroesComponent. Why haven't we marked HeroesComponent as @Injectable()?

We can add it if we really want to. It isn't necessary because the HeroesComponent is already marked with @Component, and this decorator class (like @Directive and @Pipe, which we'll learn about later) is a subtype of InjectableMetadata. It is in fact InjectableMetadata decorators that identify a class as a target for instantiation by an injector.

**Injects Logger in the application**

We're likely to need the same logger service everywhere in our application, so we put it in the project's app folder, and we register it in the providers array of the metadata for our application root component, AppComponent.

providers: [Logger]

**Implicit injector creation**:

injector = ReflectiveInjector.resolveAndCreate([Car, Engine, Tires]);

let car = injector.get(Car);

**Working with injectors directly:**

constructor(private injector: Injector) { }

car: Car = this.injector.get(Car);

heroService: HeroService = this.injector.get(HeroService);

An Injector is itself an injectable service. The component then asks the injected injector for the services it wants.Note that the services themselves are not injected into the component. They are retrieved by calling injector.get.

The get method throws an error if it can't resolve the requested service. We can call get with a second parameter (the value to return if the service is not found) instead, which we do in one case to retrieve a service (ROUS) that isn't registered with this or any ancestor injector.

## Injector providers

A provider provides the concrete, runtime version of a dependency value. The injector relies on providers to create instances of the services that the injector injects into components and other services.

We wrote the providers array like this:

providers:[Logger]

This is actually a short-hand expression for a provider registration using a provider object literal with two properties:

[{ provide:Logger, useClass:Logger}]

Occasionally we'll ask a different class to provide the service. The following code tells the injector to return a BetterLogger when something asks for the Logger.

[{ provide:Logger, useClass:BetterLogger}]

**Class provider with dependencies**

Maybe an EvenBetterLogger could display the user name in the log message. This logger gets the user from the injected UserService, which happens also to be injected at the application level.

@Injectable()

classEvenBetterLoggerextendsLogger{

constructor(private userService:UserService){super();}

log(message:string){

let name =this.userService.user.name;

super.log(`Message to ${name}: ${message}`);

}

}

Configure it like we did BetterLogger.

[UserService,{ provide:Logger, useClass:EvenBetterLogger}]

**Aliased class providers**

The dependency injector should inject that singleton instance when a component asks for either the new or the old logger. The OldLogger should be an alias for NewLogger.

We certainly do not want two different NewLogger instances in our app. Unfortunately, that's what we get if we try to alias OldLogger to NewLogger with useClass.

[NewLogger,// Not aliased! Creates two instances of `NewLogger`

{ provide:OldLogger, useClass:NewLogger}]

The solution: alias with the **useExisting** option.

[NewLogger,// Alias OldLogger w/ reference to NewLogger

{ provide:OldLogger,**useExisting**:NewLogger}]

**Value providers**

Sometimes it's easier to provide a ready-made object rather than ask the injector to create it from a class.

*// An object in the shape of the logger service***let** silentLogger = {  
**logs**: [**'Silent logger says "Shhhhh!". Provided via "useValue"'**],  
log: () => {}  
};

Then we register a provider with the useValue option, which makes this object play the logger role.

[{ provide:Logger, useValue: silentLogger }]

See more useValue examples in the [Non-class dependencies](#_Non-class_dependencies) and [OpaqueToken](#_Opaque_Token) sections.

**Factory providers**

Sometimes we need to create the dependent value dynamically, based on information we won't have until the last possible moment.

Suppose also that the injectable service has no independent access to the source of this information.

This situation calls for a **factory provider**.

A new business requirement: the HeroService must hide secret heroes from normal users. Only authorized users should see secret heroes.

**hero.service.ts**

**export class** HeroService {  
**constructor**(**private** logger:Logger, **private** isAuthroized:**boolean**) {}  
getHeroes() {  
**let** auth = **this**.isAuthroized ? **'authorized'** : **'unauthorized'**;  
**this**.logger.log(**`Getting heroes for** ${auth} **user.`**);  
**return** HEROES.filter(hero =>**this**.isAuthroized || !hero.**isSecret**);  
 }  
}

**hero.service.provider.ts**

**import** {Logger} **from "../logger.service"**;  
**import** {UserService} **from "../user.service"**;  
**import** {HeroService} **from "../hero.service"**;  
**let** *heroServiceFactory* = (logger:Logger, userService:UserService)=> {  
**return new** HeroService(logger, userService.**user**.isAuthorized);  
};  
**export let** heroServiceProvider = {  
**provide**: HeroService,  
useFactory: *heroServiceFactory*,  
**deps**: [Logger, UserService] *//deps is an array of provider tokens and injects into factory.*};

**heros.component.ts**

**import {Component} from '@angular/core';  
import {HeroListComponent} from './hero-list.component';  
import {HeroService} from "./hero.service";  
import {heroServiceProvider} from "./heroes/hero.service.provider";  
@Component({  
selector: 'my-heroes',  
template: `  
<h2>Heroes</h2>  
<hero-list></hero-list>  
 `,  
providers:[heroServiceProvider],  
directives: [HeroListComponent]  
})  
export class HeroesComponent {}**

## Dependency injection tokens

When we register a provider with an injector, we associate that provider with a dependency injection token. The injector maintains an internal token-provider map that it references when asked for a dependency. The token is the key to the map.

In all previous examples, the dependency value has been a class instance, and the class type served as its own lookup key. Here we get a HeroService directly from the injector by supplying the HeroService type as the token:

heroService:HeroService=this.injector.get(HeroService);

### Non-class dependencies

What if the dependency value isn't a class? Sometimes the thing we want to inject is a string, function, or object.

pplications often define configuration objects with lots of small facts (like the title of the application or the address of a web API endpoint) but these configuration objects aren't always instances of a class. They can be object literals such as this one:

app/app-config.ts (excerpt)

exportinterfaceAppConfig{

apiEndpoint:string;

title:string;

}

exportconst HERO\_DI\_CONFIG:AppConfig={

apiEndpoint:'api.heroes.com',

title:'Dependency Injection'

};

We can register these objects with a value provider.

TypeScript interfaces aren't valid tokens. The HERO\_DI\_CONFIG constant has an interface, AppConfig. Unfortunately, we cannot use a TypeScript interface as a token:

// FAIL! Can't use interface as provider token

[{ provide:AppConfig, useValue: HERO\_DI\_CONFIG })]

// FAIL! Can't inject using the interface as the parameter type

constructor(private config:AppConfig){}

An interface is a TypeScript design-time artifact. JavaScript doesn't have interfaces. The TypeScript interface disappears from the generated JavaScript. There is no interface type information left for Angular to find at runtime.

### OpaqueToken

One solution to choosing a provider token for non-class dependencies is to define and use an OpaqueToken. The definition looks like this:

import{OpaqueToken}from'@angular/core';

exportlet APP\_CONFIG =newOpaqueToken('app.config');

We register the dependency provider using theOpaqueToken object:

providers:[{ provide: APP\_CONFIG, useValue: HERO\_DI\_CONFIG }]

Now we can inject the configuration object into any constructor that needs it, with the help of an @Inject decorator:

constructor(@Inject(APP\_CONFIG) config:AppConfig){

this.title = config.title;

}

Although the AppConfig interface plays no role in dependency injection, it supports typing of the configuration object within the class.Or we can provide and inject the configuration object in our top-level AppComponent.

app/app.component.ts (providers)

providers:[

Logger,

UserService,

{ provide: APP\_CONFIG, useValue: HERO\_DI\_CONFIG }

]

### Optional dependencies

We can tell Angular that the dependency is optional by annotating the constructor argument with @Optional():

import{Optional}from'@angular/core';

constructor(@Optional()private logger:Logger){

if(this.logger){

this.logger.log(some\_message);

}

}

When using @Optional(), our code must be prepared for a null value. If we don't register a logger somewhere up the line, the injector will set the value of logger to null.

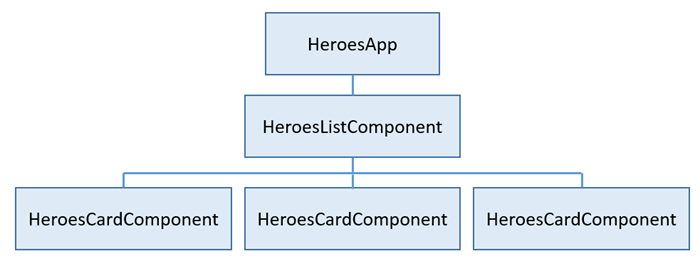
## Hierarchial Injectors

Angular has a Hierarchical Dependency Injection system. There is actually a tree of injectors that parallel an application's component tree.We can re-configure the injectors at any level of that component tree with interesting and useful results.

### Injector Tree

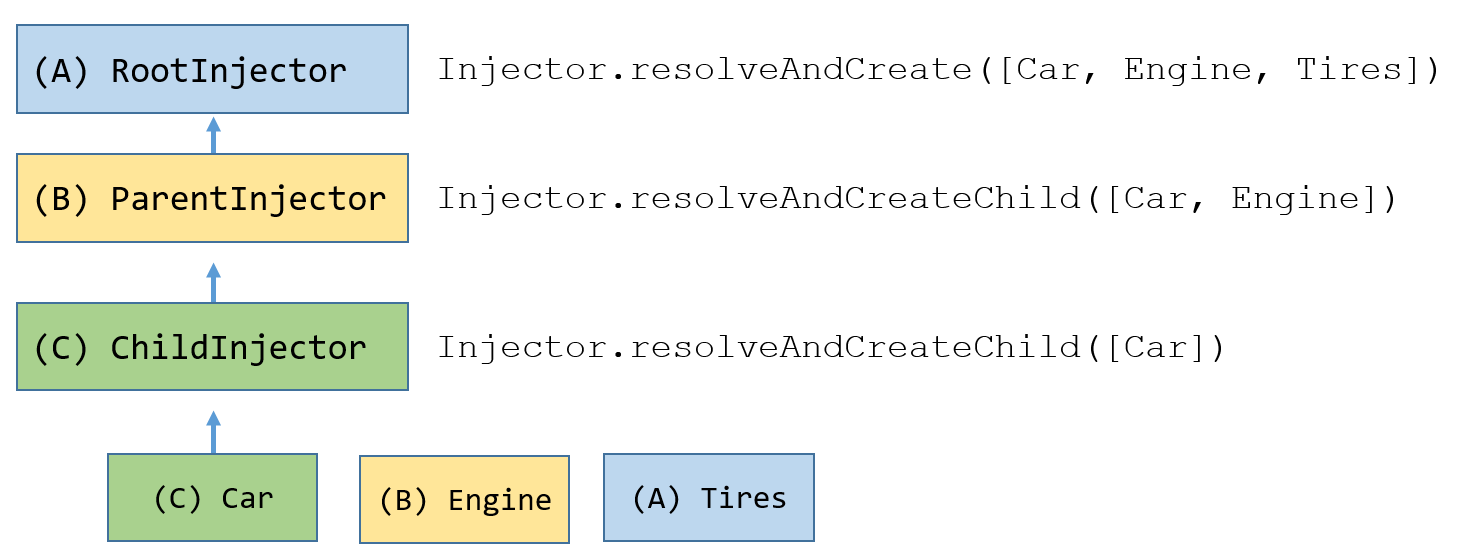
An application may have multiple injectors!

An Angular application is a tree of components. Each component instance has its own injector! (But it shares that injector with another component and there may be many different injector instances operating at different levels of the component tree.) The tree of components parallels the tree of injectors.



Each component instance gets its own injector and an injector at one level is a child injector of the injector above it in the tree.

When a component at the bottom requests a dependency, Angular tries to satisfy that dependency with a provider registered in that component's own injector. If the component's injector lacks the provider, it passes the request up to its parent component's injector. If that injector can't satisfy the request, it passes it along to its parent component's injector. The requests keep bubbling up until we find an injector that can handle the request or run out of component ancestors. If we run out of ancestors, Angular throws an error.



If we defined a RestoreService provider only on the root component, we would have exactly one instance of that service and it would be shared across the entire application.

We want each component to have its own instance of the RestoreService. Defining (or re-defining) a provider at the component level creates a new instance of the service for each new instance of that component. We've made the RestoreService a kind of "private" singleton for each HeroEditComponent, scoped to that component instance and its child components.

# Angular Modules(@NgModule)

**Angular Modules** help organize an application into cohesive blocks of functionality.

An Angular Module class is adorned with the NgModule decorator that defines metadata about the module.

This chapter explains how to create NgModule classes and how to load them, either immediately when the application launches or later, as needed, via the Router.

## Angular Modularity

Many Angular libraries are modules (e.g, FormsModule, HttpModule, RouterModule). Many third party libraries are available as Angular modules (e.g., Material Design, Ionic, AngularFire2).

Angular modules consolidate components, directives and pipes into cohesive blocks of functionality, each focused on a feature area, application business domain, workflow, or common collection of utilities.

Modules can also add services to the application. Such services might be internally-developed such as the application logger.

Modules can be loaded eagerly when the application starts. They can also be lazy loaded asynchronously by the router.

An Angular module is a class decorated with @NgModule metadata. The metadata:

* declare which **components**, **directives** and **pipes** belong together.
* make some of those classes public so that other component templates can use them.
* hide other classes as implementation details.
* import other modules with the **components**, **directives** and **pipes** it needs.
* provide services at the application level that any application component can use.

Every Angular app has at least one module class, the root module. We bootstrap that module to launch the application.

The *root module* is all we need in a simple application with a few components. As the app grows, we refactor the root module **into feature modules** that represent collections of related functionality. We then import these modules into the root module.

## AppModule -the application root Module

Every Angular app has a **root module** class. By convention it's a class called AppModule in a file named app.module.ts.

**import** {NgModule} **from '@angular/core'**;  
**import** {BrowserModule} **from '@angular/platform-browser'**;  
  
**import** {AppComponent} **from './app.component'**;  
**import** {HighlightDirective} **from "./highlight.directive"**;  
**import** {TitleComponent} **from "./title.component"**;  
**import** {UserService} **from "./user.service"**;  
  
@NgModule({  
 **imports**: [BrowserModule],  
 **declarations**: [AppComponent, HighlightDirective, TitleComponent],  
 **providers**: [UserService],  
 **bootstrap**: [AppComponent]  
})  
**export class** AppModule {  
}

This metadata imports a single helper module, BrowserModule, the module every browser app must import. BrowserModule registers critical application service providers. It also includes common directives like **NgIf** and **NgFor** which become immediately visible and usable in any of this modules component templates.

The declarations list identifies the application's only component, the root component, the top of this app's rather bare component tree.

**import** { Component } **from '@angular/core'**;  
@Component({  
 **selector**: **'my-app'**,  
 **template**: **'<h1>{{title}}</h1>'**,  
})  
**export class** AppComponent {  
 **title** = **'Minimal NgModule'**;  
}

**import** {Component, Input} **from "@angular/core"**;  
**import** {UserService} **from "./user.service"**;  
@Component({  
 **selector**: **'app-title'**,  
 **templateUrl**:**'app/title.component.html'**})  
**export class** TitleComponent {  
 @Input()  
 **subtitle** = **''**;  
 **title** = **'Angular Modules'**;  
 **user** = **''**;  
 **constructor**(userService: UserService) {  
 **this**.**user** = userService.**userName**;  
 }  
}

title.component.html:

<**h1 highlight**>{{**title**}} {{**subtitle**}}</**h1**>  
<**p \*ngIf="user"**>  
 <**i**>Welcome, {{**user**}}</**i**>  
</**p**>

**import** {Injectable} **from "@angular/core"**;  
@Injectable()  
**export class** UserService {  
 **userName** = **'Sam Spade'**;  
}

Lastly, the @NgModule.bootstrap property identifies this AppComponent as the bootstrap component. When Angular launches the app, it places the HTML rendering of AppComponent in the DOM, inside the <my-app> element tags of the **index.html**

## Bootstrapping in main.ts

We launch the application by bootstrapping the AppModule in the main.ts file.

Angular offers a variety of bootstrapping options, targeting multiple platforms. In this chapter we consider two options, both targeting the browser.

### Dynamic bootstrapping with the Just-In-Time (JIT) compiler

In the first, dynamic option, the Angular compiler compiles the application in the browser and then launches the app.

*// The browser platform with a compiler***import** { platformBrowserDynamic } **from '@angular/platform-browser-dynamic'**;  
  
*// The app module***import** { AppModule } **from './app.module'**;  
  
*// Compile and launch the module*platformBrowserDynamic().bootstrapModule(AppModule);

### Static bootstrapping with the Ahead-Of-Time (AOT) compiler

Consider the static alternative which can produce a much smaller application that launches faster, especially on mobile devices and high latency networks.

In the static option, the Angular compiler runs ahead of time as part of the build process, producing a collection of class factories in their own files. Among them is the AppModuleNgFactory.

The syntax for bootstrapping the pre-compiled AppModuleNgFactory is similar to the dynamic version that bootstraps the AppModule class.

*// The browser platform without a compiler***import** { platformBrowser } **from '@angular/platform-browser'**;  
  
*// The app module factory produced by the static offline compiler***import** { AppModuleNgFactory } **from './app.module.ngfactory'**;  
  
*// Launch with the app module factory.*platformBrowser().bootstrapModuleFactory(AppModuleNgFactory);

Because the entire application was pre-compiled, we don't ship the Angular Compiler to the browser and we don't compile in the browser.

Both the JIT and AOT compilers generate an AppModuleNgFactory class from the same AppModule source code. The JIT compiler creates that factory class on the fly, in memory, in the browser. The AOT compiler outputs the factory to a physical file that we're importing here in the static version of main.ts.

In general, the AppModule should neither know nor care how it is bootstrapped.

Many familiar Angular directives do not belong to CommonModule. For example, NgModel and RouterLink belong to Angular's FormsModule and RouterModule respectively. We must import those modules before we can use their directives.

# HTTP Client

Modern browsers support two HTTP-based APIs**: XMLHttpRequest (XHR)** and **JSONP**. A few browsers also support **Fetch**.

We use the Angular Http client to communicate with a server using a familiar HTTP request/response protocol.

## Providing HTTP Services

We'll have to register HTTP Client as a service provider with the Dependency Injection system.

**import** { HttpModule, JsonpModule } **from '@angular/http'**;  
@NgModule({  
 **imports**: [BrowserModule, FormsModule, HttpModule, JsonpModule })  
...

## RxJS Library

RxJS ("Reactive Extensions") is a 3rd party library, endorsed by Angular, that implements the asynchronous observable pattern.

The RxJS library is quite large. Size matters when we build a production application and deploy it to mobile devices. We should include only those features that we actually need. Accordingly, Angular exposes a stripped down version of Observable in the rxjs/Observable module, a version that lacks most of the operators including some we'd like to use such as the **map** method.

It's up to us to add the operators we need. We could add every RxJS operators with a single import statement. We'll put the import statements in one app/rxjs-operators.ts file.

*// See node\_module/rxjs/Rxjs.js  
// Import just the rxjs statics and operators we need for THIS app.  
  
// Statics***import 'rxjs/add/observable/throw'**;  
  
*// Operators***import 'rxjs/add/operator/catch'**;  
**import 'rxjs/add/operator/debounceTime'**;  
**import 'rxjs/add/operator/distinctUntilChanged'**;  
**import 'rxjs/add/operator/map'**;  
**import 'rxjs/add/operator/switchMap'**;  
**import 'rxjs/add/operator/toPromise'**;

Finally, we import rxjs-operatoritself in our app.component.ts:

*// Add the RxJS Observable operators we need in this app.***import './rxjs-operators'**;

The response data are in JSON string form. We must parse that string into JavaScript objects which we do by calling **response.json()**.The server we're calling always wraps JSON results in an object with a data property. We have to unwrap it to get the heroes. This is conventional web api behavior, driven by **security concerns**.

Make no assumptions about the server API. Not all servers return an object with a data property.

getHeroes(): Observable<Hero[]> {  
 **return this**.http.get(**this**.**heroesUrl**)

.**map**(**this**.extractData)

.**catch**(**this**.handleError);  
}

**private** extractData(res: Response) {  
 ***console***.log(**`response data:** ${res}**. json:** ${***JSON***.stringify(res.json())}**`**);  
 **let** body = res.json();  
 **return** body.**data** || {};  
}  
  
**private** handleError(error: **any**) {  
 *// In a real world app, we might use a remote logging infrastructure  
 // We'd also dig deeper into the error to get a better message* **let** errMsg = (error.**message**) ? error.**message** : error.**status** ? **`**${error.**status**} **-** ${error.**statusText**}**`** : **'Server error'**;  
 ***console***.error(errMsg); *// log to console instead* **return** Observable.*throw*(errMsg);  
}

Our getHeroes() could have returned the HTTP response. Bad idea! The point of a data service is to hide the server interaction details from consumers. The component that calls the HeroService wants heroes. It has no interest in what we do to get them. It doesn't care where they come from. And it certainly doesn't want to deal with a response object.

HTTP GET IS DELAYED

The http.get does not send the request just yet! This observable is cold which means the request won't go out until something subscribes to the observable. That something is the HeroListComponent.

**this**.heroService.getHeroes()  
 .subscribe(heroes => **this**.**heroes** = heroes, err =>**this**.**errorMessage** = err);

We should think of an Observable as a stream of events published by some source. We listen for events in this stream by subscribing to the Observable. In these subscriptions we specify the actions to take when the web request produces a success event (with the hero data in the event payload) or a fail event (with the error in the payload).

Whenever we deal with I/O we must be prepared for something to go wrong as it surely will. We should catch errors in the HeroService and do something with them. We may also pass an error message back to the component for presentation to the user but only if we can say something the user can understand and act upon.

<http://blog.thoughtram.io/angular/2016/01/06/taking-advantage-of-observables-in-angular2.html>

# Barrel

ay to rollup exports from several modules into a single convenience module. The barrel itself is a module file that re-exports selected exports of other modules.

*// heroes/hero.component.ts***export class** HeroComponent {}  
*// heroes/hero.model.ts***export class** Hero {}  
*// heroes/hero.service.ts***export class** HeroService {}

Without a barrel, a consumer would need three import statements:

**import** { HeroComponent } **from '../heroes/hero.component.ts'**;  
**import** { Hero } **from '../heroes/hero.model.ts'**;  
**import** { HeroService } **from '../heroes/hero.service.ts'**;

We can add a barrel to the heroes folder (called index by convention) that exports all of these items:

**export** \* **from './hero.model.ts'**; *// re-export all of its exports***export** \* **from './hero.service.ts'**; *// re-export all of its exports***export** { HeroComponent } **from './hero.component.ts'**; *// re-export the named thing*

Now a consumer can import what it needs from the barrel.

**import** { Hero, HeroService } **from '../heroes'**; *// index is implied*

# Template Syntax

# Routing

HTML5 introduced the history.pushState() and history.replaceState() methods, which allow you to add and modify history entries, respectively.

Thanks to pushState, we can make our in-app URL paths look the way we want them to look, e.g. localhost:3000/crisis-center. Our in-app URLs can be indistinguishable from server URLs.Modern HTML 5 browsers were the first to support pushState which is why many people refer to these URLs as "HTML 5 style" URLs.

We must add a <base href> element tag to the index.html to make pushState routing work. The browser also needs the base href value to prefix relative URLs when downloading and linking to css files, scripts, and images.

Add the base element just after the <head> tag. If the app folder is the application root, as it is for our application, set the href value in index.html exactly as shown here.

<basehref="/">

## Configure and add the router

**import** {RouterConfig, *provideRouter*} **from "@angular/router"**;  
**import** {HeroesComponent} **from "./heroes.component"**;  
**const** routes:RouterConfig = [{  
**path**: **'heores'**,  
**component**: HeroesComponent  
 }];  
**export const** appRouterProviders = [*provideRouter*((routes))];

The RouterConfig is an array of route definitions.

This route definition has two parts:

* **path**: the router matches this route's path to the URL in the browser address bar (/heroes).
* **component**: the component that the router should create when navigating to this route (HeroesComponent).

## Make the router available

We have to import our appRouterProviders which contains our configured router and make it available to the application by adding it to the bootstrap array.

**import** { *bootstrap* } **from '@angular/platform-browser-dynamic'**;  
**import** { AppComponent } **from './app.component'**;  
**import** {appRouterProviders} **from "./app.routes"**;  
  
*bootstrap*(AppComponent, [appRouterProviders]);

## Router Outlets and Router Links

If we paste the path, /heroes, into the browser address bar, the router should match it to the 'Heroes' route and display the HeroesComponent.We have to tell it where by adding <router-outlet> marker tags to the bottom of the template.

RouterOutlet is one of the ROUTER\_DIRECTIVES. The router displays each component immediately below the <router-outlet> as we navigate through the application.

Normally we add additional an anchor tag to the template which triggers naviation to.

template: `

<h1>{{title}}</h1>

<a [routerLink]="['/heroes']">Heroes</a>

<router-outlet></router-outlet>

`

The [routerLink] binding in the anchor tag. We bind the RouterLink directive (another of the ROUTER\_DIRECTIVES) to an array that tells the router where to navigate when the user clicks the link.

If we want the app to show the dashboard when it starts and we want to see a nice URL in the browser address bar that says /dashboard.

{

path: '',

redirectTo: '/dashboard',

pathMatch: 'full'

},

We can also configure a route with a parameter:

{

path: 'detail/:id',

component: HeroDetailComponent

},

The colon (:) in the path indicates that :id is a placeholder to be filled with a specific hero id when navigating to the HeroDetailComponent.

# Two-way Binding with NgModel

When developing data entry forms, we often want to both display a data property and update that property when the user makes changes.

The [(ngModel)] two-way data binding syntax makes that easy. Here's an example:

<input [(ngModel)]="currentHero.firstName">

[()] = BANANA IN A BOX

To remember that the parentheses go inside the brackets, visualize a banana in a box.

Alternatively, we can use the canonical prefix form:

<input bindon-ngModel="currentHero.firstName">

We could have achieved the same result with separate bindings to the <input> element's value property and input event.That’s however cumbersome.

<input [value]="currentHero.firstName" (input)="currentHero.firstName=$event.target.value" >

That ngModel directive hides these onerous details behind its own ngModel input and ngModelChange output properties.

<input [ngModel]="currentHero.firstName" (ngModelChange)="currentHero.firstName=$event">

*The ngModel input property sets the element's value property and the ngModelChange output property listens for changes to the element's value. The details are specific to each kind of element and therefore the NgModel directive only works for elements, such as the input text box, that are supported by a ControlValueAccessor. We can't apply [(ngModel)] to our custom components until we write a suitable value accessor.*

We shouldn't have to mention the data property twice. Angular should be able to capture the component’s data property and set it with a single declaration — which it can with the [( )] syntax:

<input [(ngModel)]="currentHero.firstName">

### Syntactic suger[(x)]

[(x)] is just syntactic sugar for a property binding and an event binding:

[x]="someParentProperty" (xChange)="someParentProperty=$event"

To achive the following code:

<my-comp [(myText)]="testString"></my-comp>

In the component for my-comp must have an myText property and an myTextChange property(which is a EventEmitter).

**export class** MyComp {  
 @Input() **myText**: **string**;  
 @Output() **myTextChange**: EventEmitter<**string**> = **new** EventEmitter();

*//Notify parent of changes, whenever the value of myText changes, emit an event.*onChange(newMyText:**string**) {  
**this**.**myTextChange**.emit(newMyText);  
}  
}

Another example:

In the super component:

<my-hero-detail [hero]="selectedHero" [(age)]="heroAge" />

export class HeroesComponent {private age:number=20}

In the sub component:

**<input [(ngModel)]="heroage" placeholder="0"/>**

Export class HeroDetailComponent{

@Input()  
 **heroAge**:**number**;

@Output()  
**heroageChange**:EventEmitter<**number**> = **new** EventEmitter<**number**>();  
  
onChange() {  
**this**.**ageChange**.emit(**this**.age);  
}

}

### Aliasing input/output properties

Sometimes we want the public name of an input/output property to be different from the internal name.

This is frequently the case with [attribute directives](#_Attribute_directives).Directive consumers expect to bind to the name of the directive.For example, when we apply a directive with a myClick selector to a <div> tag, we expect to bind to an event property that is also called myClick.

<div (myClick)="clickMessage=$event">click with myClick</div>

However, the directive name is often a poor choice for the name of a property within the directive class. The directive name rarely describes what the property does. The myClick directive name is not a good name for a property that emits click messages.

Fortunately, we can have a public name for the property that meets conventional expectations, while using a different name internally.

@Output('myClick') clicks =newEventEmitter<string>();// @Output(alias) propertyName = ...

Or alternative:

@Directive({

outputs:['clicks:myClick']// propertyName:alias

})

# Directives

There are three kinds of directives in Angular:

* Components
* Structural directives
* Attribute directives

A Component is really *a directive with a template*. It's the most common of the three directives and we tend to write lots of them as we build applications.

Structural directives can change the DOM layout by adding and removing DOM elements. **NgFor** and **NgIf** are two familiar examples.

An Attribute directive can change the appearance or behavior of an element. The built-in **NgStyle** directive, for example, can change several element styles at the same time.

We don't need any directive to simply set the background color. We can set it with the special Style Binding like this:

<p [*style.background*]="'lime'">I am green with envy!</p>

## Attribute directives

An attribute directive minimally requires building a controller class annotated with @Directive, which specifies the selector identifying the attribute associated with the directive. The controller class implements the desired directive behavior.

An example highlight.directive.ts:

**import** {Directive, ElementRef} **from "@angular/core"**;  
@Directive({  
**selector**: **'[myHighlight]'**})  
**export class** HighlightDirective {  
**constructor**(el: ElementRef){  
 el.**nativeElement**.**style**.**backgroundColor** = **'yellow'**;  
 }  
}

We need the ElementRef to inject into the directive's constructor so we can access the DOM element.

@Directive requires a CSS selector to identify the HTML in the template that is associated with our directive.The CSS selector for an attribute is the attribute name in square brackets. Our directive's selector is *[myHighlight]*.

We export `HighlightDirective` to make it accessible to other components.

**import** {Component} **from "@angular/core"**;  
**import** {HighlightDirective} **from "./highlight.directive"**;  
@Component({  
**selector**: **'my-app'**,  
**directives**: [HighlightDirective],  
**template**: **'<h1>My First Attribute Directive</h1><p myHighlight>Highlight me!</p>'**})

Angular creates a new instance of the directive's controller class for each matching element, injecting an Angular ElementRef into the constructor. ElementRef is a service that grants us direct access to the DOM element through its nativeElement property. That's all we need to set the element's background color using the browser DOM API.

### Respond to user action

Our directive should be able in response to a user action. We apply the @HostListener decorator to methods which are called when an event is raised.

@HostListener('mouseenter') onMouseEnter() {/\* . . . \*/}

@HostListener('mouseleave') onMouseLeave() {/\* . . . \*/}

**import** {Directive, ElementRef, HostListener} **from "@angular/core"**;  
@Directive({  
**selector**: **'[myHighlight]'**})  
**export class** HighlightDirective {  
**constructor**(**private** el: ElementRef){}  
 @HostListener(**'onmouseenter'**)  
**private** onMouseEnter(){  
**this**.hightlight(**'yellow'**);  
 }  
 @HostListener(**'onmouseleave'**)  
**private** onMouseLeave(){  
**this**.hightlight(**'null'**);  
 }  
**private** hightlight(color:**string**) {  
**this**.el.**nativeElement**.**style**.**backgroundColor** = color;  
 }  
}

### Configure the directive with binding

We should set the color externally with a binding like this:

<p [myHighlight]="color">Highlight me!</p>

We'll extend our directive class with a bindable input highlightColor property and use it when we highlight text.

**import** {Directive, ElementRef, HostListener, Input} **from "@angular/core"**;  
@Directive({  
**selector**: **'[myHighlight]'**})  
**export class** HighlightDirective {  
**private defaultColor**:**string** = **'red'**;  
**constructor**(**private** el: ElementRef){}

*//@Input(alias) alias the highlightColor property with myHighlight*  
@Input(**'myHighlight'**)**highlightColor**:**string**;  
 @HostListener(**'onmouseenter'**)  
**private** onMouseEnter(){  
**this**.hightlight(**this**.**highlightColor** || **this**.**defaultColor**);  
 }  
 @HostListener(**'onmouseleave'**)  
**private** onMouseLeave(){  
**this**.hightlight(**'null'**);  
 }  
**private** hightlight(color:**string**) {  
**this**.el.**nativeElement**.**style**.**backgroundColor** = color;  
 }  
}

Update the template:

<h1>My First Attribute Directive</h1>

<h4>Pick a highlight color</h4>

<div>

<inputtype="radio"name="colors" (click)="color='lightgreen'">Green

<inputtype="radio"name="colors" (click)="color='yellow'">Yellow

<inputtype="radio"name="colors" (click)="color='cyan'">Cyan

</div>

<p [myHighlight]="color">Highlight me!</p>

### Bind to a second property

Let's allow the template developer to set the default color, the color that prevails until the user picks a highlight color.We'll add a second input property to HighlightDirective called defaultColor:

@Input() **set** defaultColor(colorName: **string**){  
**this**.**defaultColor** = colorName || **this**.**defaultColor**;  
}

In html-template:

<p [myHighlight]="color" [defaultColor]="'violet'">

Highlight me too!

</p>

# Animations

**import** {Component, *trigger*, *state*, *style*, *transition*, *animate*, Input} **from "@angular/core"**;  
**import** {Heroes} **from "./hero.service"**;  
  
@Component({  
**moduleId**: ***module***.**id**,  
**selector**: **'hero-list-basic'**,  
**template**: **`  
<ul>  
<li \*ngFor="let hero of heroes" @heroState="hero.state" (click)="hero.toggleState()">  
 {{hero.name}}  
</li>  
</ul>  
`**,  
**styleUrls**: [**'hero-list.component.css'**],  
**animations**: [  
*trigger*(**'heroState'**, [  
*state*(**'inactive'**, *style*({  
**backgroundColor**: **'#eee'**,  
**transform**: **'scale(1)'**})),  
*state*(**'active'**, *style*({  
**backgroundColor**:**'#cfd8dc'**,  
**transform**:**'scale(1.1)'**})),  
*transition*(**'inactive => active'**, *animate*(**'100ms ease-in'**)),  
*transition*(**'active => inactive'**, *animate*(**'100ms ease-out'**))  
 ])  
 ]  
})  
**export class** HeroListBasicComponent {  
 @Input() **heroes**:Heroes;  
**constructor**(){}  
}

We define an animation trigger called heroState in attribut animations. We can use this animation by attaching it to one or more elements in the compoment's template using the "@triggername" syntax.

Here we've applied the animation trigger to every element repeated by an ngFor. Each of the repeated elements will animate independently. We're binding the value of the attribute to the expression *hero.state*.We expect it to always be either inactive or active, since that's what we have defined animation states for.an animated transition is shown whenever a hero object changes state!

## States and Transitions

Angular animations are defined in terms of logical **states** and **transitions** between states.

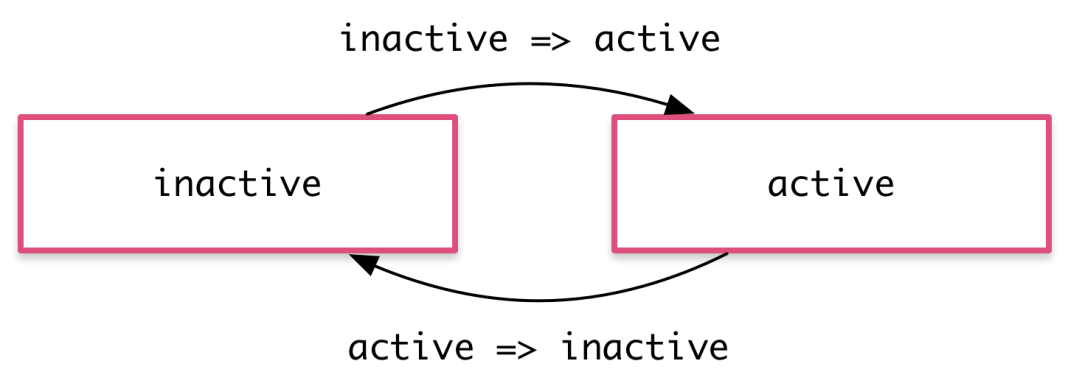
An animation state is a string value that we define in our application code. In the example above we used the states **'active'** and **'inactive'** based on the logical state of hero objects.

The source of the state can be a simple object attribute as it was in this case, or it can be a value computed in a method. The important thing is that we can read it into the component's template.

These state definitions specify the end styles of each state. They are applied to the element once it has transitioned to that state, and will stay as long as it remains in that state.

Once we have states, we can define transitions between the states. Each transition controls the timing of switching between one set of styles and the next:

*transition*(**'inactive => active'**, *animate*(**'100ms ease-in'**)),  
*transition*(**'active => inactive'**, *animate*(**'100ms ease-out'**))



If we have the same timing configuration for several transitions, we can combine them into the same transition definition:

transition('inactive => active, active => inactive', animate('100ms ease-out'))

When we have the same timing for both directions of a transition, as we do in the previous example, we can use the <=> shorthand syntax:

transition('inactive <=> active', animate('100ms ease-out'))

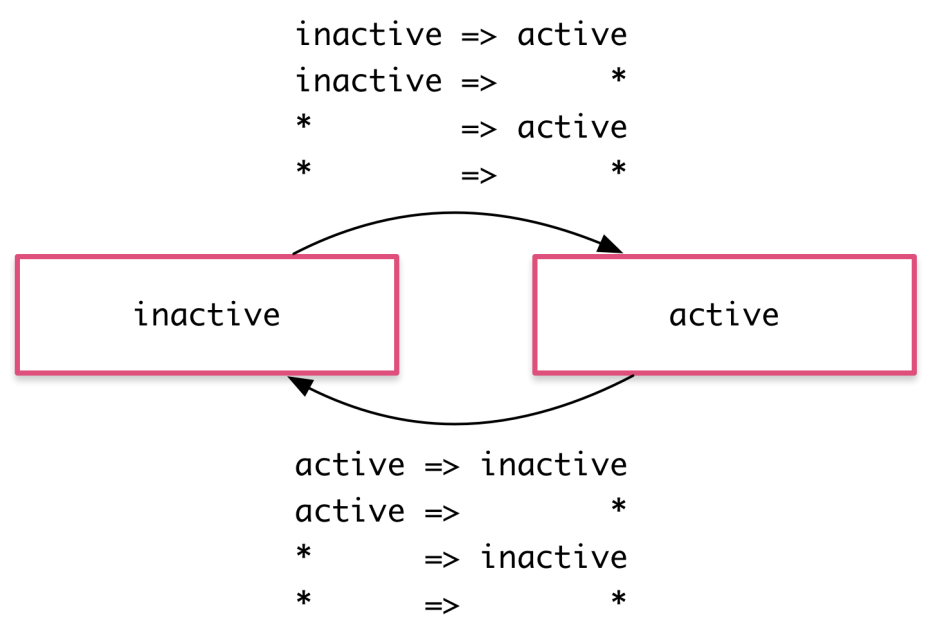
Sometimes we have styles that we want to apply during an animation but not keep around after it finishes.We can define such styles inline in the **transition**.

*transition*(**'inactive => active'**, [  
*// start style  
style*({  
**backgroundColor**: **'#000'**,  
**transform**: **'scale(2)'**}),  
*animate*(**'1000ms ease-in'**,  
*// Animation from start style to this style, change to style for active state  
style*({  
**backgroundColor**: **'#fff'**,  
**transform**: **'scale(5)'**}))  
]),

### The wildcard state \*

The \* ("wildcard") state matches any animation state. This is useful for defining styles and transitions that should apply regardless of which state the animation is in. For example:

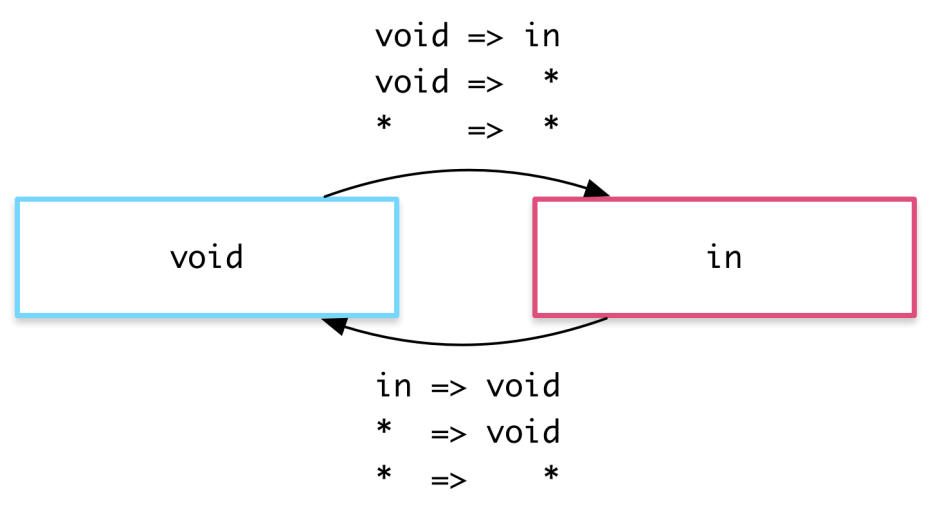
* The active => \* transition applies when the element's state changes from active to anything else.
* The \* => \* transition applies when any change between two states takes place.



### The void state

There's one special state called void that may apply to any animation. It applies when the element is not attached to a view. This may be because it has not yet been added or because it has been removed. The void state is useful for defining "enter" and "leave" animations.

For example the \* => void transition applies when the element leaves the view, regardless of what state it was in before it left.



The wildcard state \* also matches void.

## Example: Entering and Leaving

Using the void and \* states we can define transitions that animate the entering and leaving of elements:

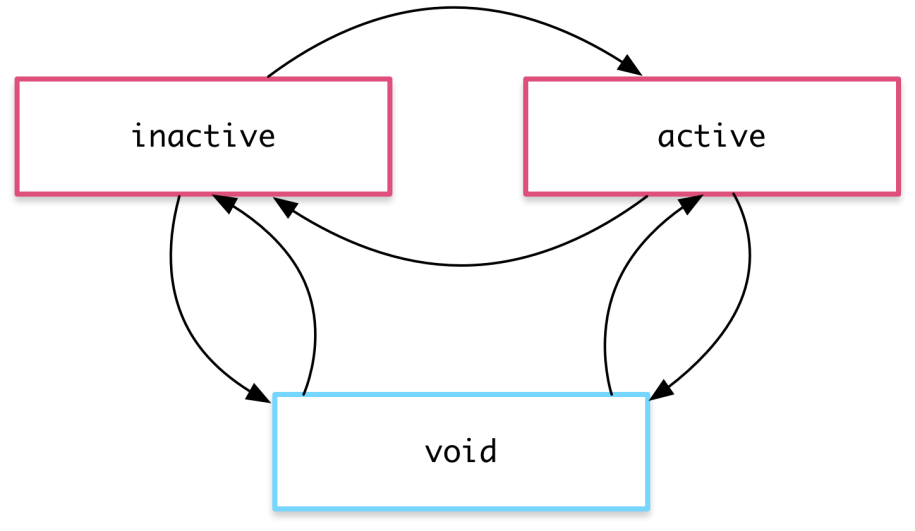
* Enter: void => \*
* Leave: \* => void

**animations**: [  
*trigger*(**'flyInOut'**, [  
*state*(**'in'**, *style*({**transform**: **'translateX(0)'**})),  
*transition*(**'void => \*'**, [  
*style*({**transform**: **'translateX(-100%)'**}),  
*animate*(100)  
 ]),  
*transition*(**'\* => void'**, [  
*animate*(100, *style*({**transform**: **'translateX(100%)'**}))  
 ])  
 ])  
]

Note that in this case we have the styles applied to the void state directly in the transition definitions, and not in a separate state(void) definition. We do this because we want the transforms to be different on enter and leave: The element enters from the left and leaves to the right.

### Combine with different state

We now have fine-grained control over each transition:



**animations**: [  
*trigger*(**'heroState'**, [  
*state*(**'inactive'**, *style*({  
**transform**: **'translateX(0) scale(1)'**})),  
*state*(**'active'**, *style*({  
**transform**: **'translateX(0) scale(1.1)'**})),  
*transition*(**'inactive => active'**, *animate*(**'100ms ease-in'**)),  
*transition*(**'active => inactive'**, *animate*(**'100ms ease-out'**)),  
*transition*(**'void => inactive'**, [  
*style*({**transform**: **'translateX(-100%) scale(1)'**}),  
*animate*(1000)  
 ]),  
*transition*(**'inactive => void'**, [  
*animate*(1000, *style*({**transform**: **'translateX(100%) scale(1)'**}))  
 ]),  
*transition*(**'void => active'**, [  
*style*({**transform**: **'translateX(0) scale(0)'**}),  
*animate*(2000)  
 ]),  
*transition*(**'active => void'**, [  
*animate*(2000, *style*({**transform**: **'translateX(0) scale(0)'**}))  
 ])  
 ])  
]

## Animatable Properties and Units

Since Angular's animation support builds on top of Web Animations, we can animate any property that the browser considers animatable.This includes positions, sizes, transforms, colors, borders and many others. The W3C maintains [a list of animatable properties](https://www.w3.org/TR/css3-transitions/#animatable-properties).

For positional properties that have a numeric value, we can define a unit by providing the value as a string with the appropriate suffix:

* '50px'
* '3em'
* '100%'

For most dimensional properties we can also just define a number which is then assumed to be in pixels:

* 50 is the same as saying '50px'

## Automatic Property Calculation

Sometimes the value of a dimensional style property that we want to animate is not known until at runtime.or example, it is quite common for elements to have widths and heights that depend on their content and the screen size. These properties are often tricky to animate with CSS.

With Angular we can use a special **\*** property value in these cases. What it means is that the value of this property will be computed at runtime and then plugged into the animation.

**animations**: [  
 trigger(**'shrinkOut'**, [  
 state(**'in'**, style({**height**: **'\*'**})),  
 transition(**'\* => void'**, [  
 style({**height**: **'\*'**}),  
 animate(250, style({**height**: 0}))  
 ])  
 ])  
]

## Animation Timing

There are three timing properties we can tune for every animated transition:

* The duration
* the delay,
* the easing function

### Duration

The duration controls how long the animation takes to run from start to finish. We can define a duration in three ways:

* As a plain number, in milliseconds: 100
* In a string, as milliseconds: '100ms'
* In a string, as seconds: '0.1s'

### Delay

The delay controls how long to wait after an animation triggers before the transition actually begins. We can define one by adding it in the same string following the duration. It also has the same format options as the duration:

Wait for 100ms and then run for 200ms: '0.2s 100ms'

### Easing

The easing function controls how the animation accelerates and decelerates during its runtime. For example, using an **ease-in** function means the animation begins relatively slowly but then picks up speed as it progresses. We can control the easing by adding it as a third value in the string after the duration and the delay (or as the second value when there is no delay):

Wait for 100ms and then run for 200ms, with easing: '0.2s 100ms ease-out'

Run for 200ms, with easing: '0.2s ease-in-out'

**animations**: [  
 trigger(**'flyInOut'**, [  
 state(**'in'**, style({**opacity**: 1, **transform**: **'translateX(0)'**})),  
 transition(**'void => \*'**, [  
 style({  
**opacity**: 0,  
**transform**: **'translateX(-100%)'**}),  
 animate(**'0.2s ease-in'**)  
 ]),  
 transition(**'\* => void'**, [  
 animate(**'0.2s 10 ease-out'**, style({  
**opacity**: 0,  
**transform**: **'translateX(100%)'**}))  
 ])  
 ])  
]

Here are a couple of custom timings in action. Both "enter" and "leave" last for 200 milliseconds but they have different easings. The leave begins after a slight delay.

### Multi-Step Animations with keyframes

With animation keyframes we can go beyond a simple transition between two sets of styles to a more intricate animation that goes through one or more intermediate styles in between.

For each keyframe, we can specify an offset that defines at which point in the animation that keyframe applies. The offset is a number between zero, which marks the beginning of the animation, and one, which marks the end.

animations: [  
 trigger(**'flyInOut'**, [  
 state(**'in'**, style({**transform**: **'translateX(0)'**})),  
 transition(**'void => \*'**, [  
 animate(300, keyframes([  
 style({**opacity**: 0, **transform**: **'translateX(-100%)'**, **offset**: 0}),  
 style({**opacity**: 1, **transform**: **'translateX(15px)'**, **offset**: 0.3}),  
 style({**opacity**: 1, **transform**: **'translateX(0)'**, **offset**: 1.0})  
 ]))  
 ]),  
 transition(**'\* => void'**, [  
 animate(300, keyframes([  
 style({**opacity**: 1, **transform**: **'translateX(0)'**, **offset**: 0}),  
 style({**opacity**: 1, **transform**: **'translateX(-15px)'**, **offset**: 0.7}),  
 style({**opacity**: 0, **transform**: **'translateX(100%)'**, **offset**: 1.0})  
 ]))  
 ])  
 ])  
]

In this example we add some "bounce" to our enter and leave animations with keyframes.

Note that the offsets are not defined in terms of absolute time. They are relative measures from 0 to 1. The final timeline of the animation will based on the combination of keyframe offsets, duration, delay, and easing.

Defining offsets for keyframes is optional. If we omit them, offsets with even spacing are automatically assigned. For example, three keyframes without predefined offsets will receive offsets 0, 0.5, and 1.

### Parallel Animation Groups

we may also want to configure different timings for animations that happen in parallel. For example, we may want to animate two CSS properties but use a different easing function for each one.

For this we can use animation groups. In this example we use groups both on enter and leave so that we can use two different timing configurations. Both are applied to the same element in parallel, but run independent of each other:

**animations**: [  
 trigger(**'flyInOut'**, [  
 state(**'in'**, style({**width**: 120, **transform**: **'translateX(0)'**, **opacity**: 1})),  
 transition(**'void => \*'**, [  
 style({**width**: 10, **transform**: **'translateX(50px)'**, **opacity**: 0}),  
 group([  
 animate(**'0.3s 0.1s ease'**, style({  
**transform**: **'translateX(0)'**,  
**width**: 120  
})),  
 animate(**'0.3s ease'**, style({  
**opacity**: 1  
}))  
 ])  
 ]),  
 transition(**'\* => void'**, [  
 group([  
 animate(**'0.3s ease'**, style({  
**transform**: **'translateX(50px)'**,  
**width**: 10  
})),  
 animate(**'0.3s 0.2s ease'**, style({  
**opacity**: 0  
}))  
 ])  
 ])  
 ])  
]

One group animates the element transform and width. The other animates the opacity.

# Component Styles

For every Angular 2 component we write, we may define not only an HTML template, but also the CSS styles that go with that template, specifying any selectors, rules, and media queries that we need.

One way to do this is to set the styles property in the component metadata. The styles property takes an array of strings that contain CSS code.

@Component({  
**selector**: **'hero-app'**,  
**template**: **`  
<h1>Tour of Heroes</h1>  
<hero-app-main [hero]=hero></hero-app-main>`**,  
**styles**: [**'h1 { font-weight: normal; }'**]  
})  
**export class** HeroAppComponent {  
*/\* . . . \*/*}

Component styles differ from traditional, global styles:

The selectors we put into a component's styles only apply within the template of that component. The h1 selector in the example above only applies to the <h1> tag in the template of HeroAppComponent. Any <h1> elements elsewhere in the application are unaffected.

## Special Selectors

Component styles have a few special selectors.

### :host

Use the :host pseudo-class selector to target styles in the element that hosts the component (as opposed to targeting elements inside the component's template).This is the only way we can target the host element.

**styles**:[**':host {display: block;border: 1px solid black;}'**]

In the next example we target the host element again, but only when it also has the active CSS class.

**styles**:[**`:host(.active) {  
 border-width: 3px;  
}`**]

### :host-context

Sometimes it is useful to apply styles based on some condition outside a component's view. For example, there may be a CSS theme class applied to the document <body> element, and we want to change how our component looks based on that.

In the following example, we apply a background-color style to all <h2> elements inside the component, only if some ancestor element has the CSS class theme-light.

**:host-context(.theme-light) h2 {  
 background-color: #eef;  
}**

### /deep/ (Alias >>>)

Component styles normally apply only to the HTML in the component's own template.

We can use the /deep/ selector to force a style down through the child component tree into all the child component views. The /deep/ selector works to any depth of nested components, and it applies both to the view children and the content children of the component.

In this example, we target all <h3> elements, from the host element down through this component to all of its child elements in the DOM:

**:host /deep/ h3 {  
 font-style: italic;  
}**

The /deep/ selector also has the alias >>>. We can use either of the two interchangeably.

The /deep/ and >>> selectors should only be used with emulated view encapsulation. This is the default and it is what we use most of the time.

## Loading Styles into Components

* Add a styles array(usually just one string) property to the @Component decorator.
* Add <style> tags to the HTML Template.
* Add a styleUrls attribute into the @Component decorator. Relative paht to the app root.
* Add <link> tags to the HTML Template. As with styleUrls, href is also the relative path to the application root.
* Import CSS files into our CSS files by using the standard CSS @import rule. In this case the URL is relative to the CSS file into which we are importing.

@Component({  
**selector**: **'hero-controls'**,  
**template**: **`  
<style>  
button {  
 background-color: white;  
 border: 1px solid #777;  
 }  
</style>  
<h3>Controls</h3>  
<button (click)="activate()">Activate</button>  
 `**})

@Component({  
**selector**: **'hero-team'**,  
**template**: **`  
<link rel="stylesheet" href="app/hero-team.component.css">  
<h3>Team</h3>  
<ul>  
<li \*ngFor="let member of hero.team">  
 {{member}}  
</li>  
</ul>`**})

## Controlling View Encapsulation

component CSS styles are encapsulated into the component's own view and do not affect the rest of the application.

We can control how this encapsulation happens on a per component basis by setting the view encapsulation mode in the component metadata. There are three modes to choose from:

* Native: uses the browser's native [**Shadow DOM**](https://developer.mozilla.org/en-US/docs/Web/Web_Components/Shadow_DOM) implementation to attach a Shadow DOM to the component's host element, and then puts the component view inside that Shadow DOM. The component's styles are included within the Shadow DOM.
* Emulated (default): emulates the behavior of Shadow DOM by preprocessing (and renaming) the CSS code to effectively scope the CSS to the component's view.
* None: Angular does no view encapsulation. Angular adds the CSS to the global styles.This is essentially the same as pasting the component's styles into the HTML.

Setting example:

*// warning: few browsers support shadow DOM encapsulation at this time***encapsulation**: ViewEncapsulation.*Native*

# Ahead-of-Time(AoT) Compilation in Angular 2

## Why compilation

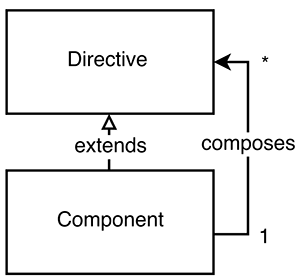
The short answer of this question is - *We need compilation for achieving higher level of efficiency of our Angular applications.* By efficiency I mean performance, energy and bandwidth consumption.

# ViewChildren and ContentChildren

We will take a look at how we can pass access these two different kinds of children from their parent component. Along the content we are also going to mention what the difference between the properties providers and viewProviders of the @Component decorator is.

A typical design pattern for developing user interface is the composite pattern. It allows us to compose different primitives and treat them the same way as a single instance.

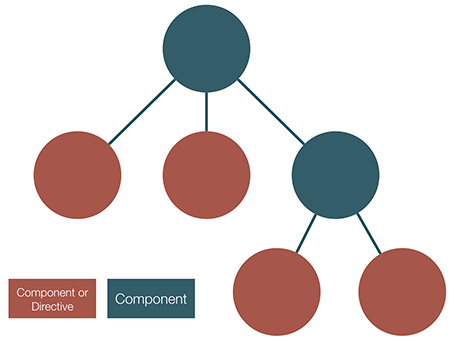
We can illustrate this graphically by the following structural diagram:



In the figure above we have two elements:

* Directive - A self-contained element which holds some logic, but does not contain any structure.
* Component - An element, which specifies the Directive element and holds a list of other Directive instances (which could also be components since Component extends Directive).

This means that using the preceding abstractions we can build structures of the following form:



The leaf elements on the diagram are either directives or components that don’t hold references.

## Composition of Components in Angular 2

In order to illustrate the concept we build a simple application:

@Component({  
 **selector**: **'todo-app'**,  
 **providers**: [TodoList],  
 **template**: **`  
 <section>  
 Add todo:  
 <todo-input (todo)="addTodo($event)"></todo-input>  
 </section>  
 <section>  
 <h4 \*ngIf="todos.getAll().length">Todo list</h4>  
 <todo \*ngFor="let todo of todos.getAll()" [todo]="todo">  
 </todo>  
 </section>  
 <ng-content select="footer"></ng-content>  
 `**})  
**class** TodoAppComponent {  
 **constructor**(**private** todos: TodoList) {}  
 addTodo(todo) {  
 **this**.todos.add(todo);  
 }  
}

We can use the component in the following way:

<todo-app></todo-app>

Between the opening and closing tags of the todo-app element we can put some content:

<todo-app>  
<footer>  
 Yet another todo app!  
</footer>  
</todo-app>

Notice the last element in its template:

<ng-content select="footer"></ng-content>.

With ng-content we can grab the content between the opening and closing tag of the todo-app element and project it somewhere inside of the template! The value of the select attribute is a CSS selector, which allows us to select the content that we want to project. For instance in the example above, the footer will be injected at the bottom of the rendered todo component.

We can also skip the select attribute of the ng-content element. In this case we will project the entire content passed between the opening and closing tags on the place of the ng-content element.

## What are ViewChildren and ContentChildren?

* **ViewChildren**: The children element which are located inside of its template of a component.
* **ContentChildren**: elements which are used between the opening and closing tags of the host element of a given component.

Angular 2 provides the following property decorators in the @angular/core package: @ViewChildren, @ViewChild, @ContentChildren and @ContentChild.

### Accessing View Children

We can use them the following way:

**import** {ViewChild, ViewChildren, Component, AfterViewInit...} from **'@angular/core'**;  
*// ...*@Component({  
 **selector**: **'todo-app'**,  
 **providers**: [TodoList],  
 **template**: **`...`**})  
**class** TodoAppComponent **implements** AfterViewInit {  
 @ViewChild(TodoInputComponent) **inputComponent**: TodoInputComponent  
 @ViewChildren(TodoComponent) **todoComponents**: QueryList<TodoComponent>;  
  
 **constructor**(**private** todos: TodoList) {}  
 ngAfterViewInit() {  
 *// available here* }  
}  
*// ...*

The example above shows how we can take advantage of @ViewChildren and @ViewChild. Basically we can decorate a property and this way query the view of an element. In the example above, we query the TodoInputComponent child component with @ViewChild and TodoComponent with @ViewChildren. We use different decorators since we have only a single input, so we can grab it with @ViewChild but we have multiple todo items rendered, so for them we need to apply the @ViewChildren decorator.

Another thing to notice are the types of the inputComponent and todoComponents properties. The first property is of type TodoInputComponent. It’s value can be either null if Angular haven’t found such child or reference to the instance of the component’s controller (in this case, reference to an instance of the TodoInputComponent class). On the other hand, since we have multiple TodoComponent instances which can be dynamically added and removed from the view, the type of the todoComponents property is QueryList<TodoComponent>. We can think of the QueryList as an observable collection, which can throw events once items are added or removed from it.

Since Angular’s DOM compiler will process the todo-app component before its children, during the instantiation of the todo-app component the inputComponent and todosComponent properties will not be initialized. Their values are going to be set in the **ngAfterViewInit** life-cycle hook.

### Accessing Content Children

Almost the same rules are valid for the element’s content children, however, there are some slight differences.

@Component({  
 **selector**: **'footer'**,  
 **template**: **'<ng-content></ng-content>'**})  
**class** Footer {}  
  
@Component(...)  
**class** TodoAppComponent {...}  
  
@Component({  
 **selector**: **'app'**,  
 **styles**: [  
 **'todo-app { margin-top: 20px; margin-left: 20px; }'** ],  
 **template**: **`  
 <content>  
 <todo-app>  
 <footer>  
 <small>Yet another todo app!</small>  
 </footer>  
 </todo-app>  
 </content>  
 `**})  
**export class** AppComponent {}

Footer is a content child. We can access it in the following way:

*// ...*@Component(...)  
**class** TodoAppComponent **implements** AfterContentInit {  
 @ContentChild(Footer) **footer**: Footer;  
 ngAfterContentInit() {  
 *// this.footer is now with value set* }  
}  
*// ...*

As we can see from above the only two differences between accessing view children and content children are **the decorators** and **the life-cycle hooks**.

## viewProviders vs providers (deprecated?)

As final step lets see what the difference between providers and viewProviders is.

**class** TodoList {  
 **private todos**: Todo[] = [];  
 add(todo: Todo) {}  
 remove(todo: Todo) {}  
 set(todo: Todo, index: **number**) {}  
 get(index: **number**) {}  
 getAll() {}  
}  
  
@Component({  
 *// ...* **viewProviders**: [TodoList],  
 *// ...*})  
**class** TodoAppComponent {  
 **constructor**(**private** todos: TodoList) {}  
 *// ...*}

Inside of the @Component decorator we set the viewProviders property to an array with a single element - the TodoList service.

Providers declared in given component with **viewProviders** are accessible by the component itself and all of its view successors. But it cannot be accessed by the content children.

If we try to inject this service in Footer component’s constructor we are going to get the following runtime error:

EXCEPTION: No provider for TodoList! (Footer -> TodoList)

# Funktionsweise Change Detection

Application state change is caused by three factors:

* **Events**: User events like: click, change, input, submit,…
* **XMLHttpRequest**: When fetching data from a remote Service
* **Timers()**: setTimeout(), setInterval()

All of them are **asynchronous**.

Angular takes advantage of Zones. Zones monkey-patches global asynchronous operations such as setTimeout() and addEventListener(), which is why Angular can easily find out, when to update the DOM.

ObservableWrapper.subscribe(**this**.zone.onTurnDone, () => {  
 **this**.**zone**.run(() => {  
 **this**.tick();  
 });  
});  
  
tick() {  
 *// perform change detection* **this**.changeDetectorRefs.forEach((detector) => {  
 detector.detectChanges();  
 });  
}

Whenever Angular’s zone emits an onTurnDone event, it runs a task that performs change detection for the entire application.

The onTrunDone event is not part of the default Zone API. It turns out that Angular introduces its own zone called NgZone.

NgZone is basically a forked zone that extends its API and adds some additional functionality to its execution context.

**var** myZone = zone.fork();  
myZone.run(main);

This really just gives us a new zone with the same power of the original zone. Hooks are defined using a ZoneSpecification that we can pass to fork(). We can take advantage of the following hooks:

* **onZoneCreated** - Runs when zone is forked
* **beforeTask** - Runs before a function called with zone.run is executed
* **afterTask** - Runs after a function in the zone runs
* **onError** - Runs when a function passed to zone.run will throw

**var** myZoneSpec = {  
 beforeTask: **function** () {  
 ***console***.log(**'Before task'**);  
 },  
 afterTask: **function** () {  
 ***console***.log(**'After task'**);  
 }  
};  
  
**var** myZone = zone.fork(myZoneSpec);  
myZone.run(main);

NgZone adds to the API is the following set of custom events we can subscribe to, as they are observable streams:

* onTurnStart() - Notifies subscribers just before Angular’s event turn starts. Emits an event once per browser task that is handled by Angular.
* onTurnDone() - Notifies subscribers immediately after Angular’s zone is done processing the current turn and any micro tasks scheduled from that turn.
* onEventDone() - Notifies subscribers immediately after the final onTurnDone() callback before ending VM event. Useful for testing to validate application state.

**Running code outside Angular’s Zone**

Since NgZone is really just a fork of the global zone, Angular has full control over when to run something inside its zone to perform change detection and when not. Why is that useful? Well, it turns out that we don’t always want Angular to magically perform change detection.

That’s why NgZone comes with an API runOutsideAngular() which performs a given task in NgZone’s parent zone, which does not emit an onTurnDone event, hence no change detection is performed.

**Running code inside Angular’s Zone**

using zone.run() which in turn causes Angular to perform change detection which will update the view.

# Best practices

* This is a golden rule: always delegate data access to a supporting service class.
* Components are easier to test and debug when their constructors are simple and all real work (especially calling a remote server) is handled in a separate method.

# Words

**cumbersome**英 **['kʌmbəsəm]**美 **['kʌmbərsəm]****adj.笨重的；不方便的**

**cumber**英 ['kʌmbə] 美 ['kʌmbə]v.**拖累；妨碍**n.**妨碍；累赘**

**onerous**英 ['əʊnərəs] 美 ['ɑːnərəs]adj.**繁重的**

# consolidate英 [kən'sɒlɪdeɪt] 美 [kən'sɒləˌdeɪt]v.**合并；统一；巩固**